[redacted]

510.84
Il6r
no.593

Math

UIUCDCS-R-73-593

DESIGN OF TOTALLY SELF-CHECKING ASYNCHRONOUS SEQUENTIAL MACHINES

by

DANIEL AVERY PITT

September 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-73-593

DESIGN OF TOTALLY SELF-CHECKING ASYNCHRONOUS SEQUENTIAL MACHINES

By

DANIEL AVERY PITT

B.S., Duke University, 1971

Urbana, Illinois

*Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1973.

ACKNOWLEDGMENT

I wish to thank my adviser, Dr. Gernot Metze, Professor of Electrical Engineering and Research Professor at the Coordinated Science Laboratory, for his guidance, thought, and assistance in the preparation of this thesis. His foresight in asking all the right questions and his ability to project technical problems in their proper perspective are greatly appreciated.

In addition, I would like to express special gratitude to my father, Hy Pitt, for supplying me with most of my motivation and encouragement and without whose example of earnestness I would probably have achieved far less than I have.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

A totally self-checking asynchronous sequential machine is one which detects and indicates its own internal faults in real time. Methods have been developed, most notably by Maki _et al_. [1], for designing asynchronous machines that indicate faults by entering illegal states. To detect these faults, one must use some sort of combinational circuit to determine if the present state, represented by the binary vector of the state variables, is a valid one. Faults in this combinational circuitry can cause illegal states to be interpreted as valid, thus masking the error in the combinational network as well as the one which produced the erroneous state. The value of having a totally self-checking network is that faults in this combinational portion are indicated, as well as faults in the sequential portion. Thus erroneous information can never be passed on to succeeding networks.

The method presented herein is that of designing a totally self-checking network consisting of two portions, a sequential machine and a checker, each of which is (and must be) totally self-checking. For any network to be self-checking both its inputs and outputs must be encoded so that it can distinguish between valid and invalid codewords. For the sequential portion the encoded input will be the present state and the output will be the next state. This encoded output will be the encoded input to the checker, whose output is a simple 1-out-of-2 code.

1. TOTALLY SELF-CHECKING MODEL

A totally self-checking circuit is defined by Anderson [2] according to the way in which it maps inputs into outputs, as follows:

Definition: A circuit is self-testing if, for every fault from a prescribed set, the circuit produces a non-code output for at least one code input.

Definition: A circuit is fault-secure if, for every fault from a prescribed set, the circuit never produces an incorrect code output for code inputs.

Definition: A circuit is totally self-checking if it is both self-testing and fault secure.

The set of faults which is considered here is the set of all single faults on gate inputs or outputs. The fault model is that of a line being either stuck-at-one or stuck-at-zero, with the fault being either transient or solid (permanent). A totally self-checking check circuit is also code-disjoint, in that when there are no internal faults it always maps non-codewords into non-codewords and codewords into codewords.

2. OPERATION OF SELF-CHECKING CHECKERS

Self-checking check circuits have been investigated by Anderson and Metze [3] and operate by mapping input codewords onto a 1-out-of-2 output code, where the output functions, denoted as f and g, are independently implemented from the inputs. Thus valid input codewords result in outputs of fg = 10 or fg = 01, while non-codeword inputs are mapped onto fg = 00 or fg = 11. Furthermore, single or unidirectional faults in the checker also result in outputs of fg = 00 or fg = 11. Monitoring these outputs is required to initiate the repair process in the case that a fault occurs. It is assumed that this monitoring is done so that each fault is repaired before another has a chance to occur. Methods of monitoring the checker are discussed by Anderson [2].

Self-checking checkers have not yet been designed to check arbitrary input codes, but have been developed for parity codes and certain classes of m-out-of-n codes. In the latter case, the most clearly stated techniques are for the case where n = 2m. Codes of this type are called k-out-of-2k codes, and it is this kind of code which will be used in this paper. Certain m-out-of-n codes can be converted to k-out-of-2k codes using a self-checking code-disjoint translator, but designing self-checking asynchronous machines to yield these special cases in their state assignments is much more difficult than presenting a general design algorithm which yields a k-out-of-2k state assignment, where k may be any integral value.

As the self-testing property of totally self-checking checkers implies, every internal fault has at least one corresponding input codeword which, when applied to the checker in the presence of that fault, yields a non-code output, namely fg = 00 or fg = 11. Thus the code space of all the input codewords is sufficient to check the checker for the presence of all internal faults. The size of this code space is the number of binary words of length 2k in which k bits are ones, or $\binom{2k}{k}$. As will be shown later, checkers with certain internal structure may require fewer than $\binom{2k}{k}$ codewords.

3. DESIGN OF TOTALLY SELF-CHECKING CHECKERS FOR k-OUT-OF-2k CODES

This design method presented by Anderson and Metze [3] requires dividing up the input bits into two groups A and B of k bits each, denoted by $A = \{a_1, \cdots, a_k\}$, $B = \{b_1, \cdots, b_k\}$. Thus a word in a 4-out-of-8 code would be represented by $a_1a_2a_3a_4b_1b_2b_3b_4$. Let k_a and k_b denote the number of ones occurring in each group, respectively. Define the majority function $T(k_a \geq i)$ to have the value 1 if and only if the condition inside the parentheses is true, in this case if group A has at least i ones. If i = 1 for example, then $T(k_a \geq i) = (a_1+a_2+\cdots+a_k)$. The two output functions f_k and g_k are given by

$$f_k = \sum_{\substack{i=0 \\ i \text{ odd}}}^{k} T(k_a \geq i)\cdot T(k_b \geq k-i)$$

$$g_k = \sum_{\substack{i=0 \\ i \text{ even}}}^{k} T(k_a \geq i)\cdot T(k_b \geq k-i) .$$

Note: "+" denotes the OR function; "·" denotes AND.

Example: for k = 3

$$f_3 = T(k_a \geq 1)\cdot T(k_b \geq 2) + T(k_a \geq 3)\cdot T(k_b \geq 0)$$

$$= (a_1+a_2+a_3)(b_1b_2+b_1b_3+b_2b_3) + (a_1a_2a_3)\cdot 1$$

$$g_3 = T(k_a \geq 0)\cdot T(k_b \geq 3) + T(k_a \geq 2)\cdot T(k_b \geq 1)$$

$$= 1\cdot (b_1b_2b_3) + (a_1a_2+a_1a_3+a_2a_3)(b_1+b_2+b_3) .$$

Each output function must be independently implemented from the inputs and may not share any logic with the other. Since f and g are both positive functions, any number of decreasing (one-to-zero) faults in the input code or in the checker results in fg = 00 and any number of increasing (zero-to-one) faults results in fg = 11.

4. DESIGN OF SELF-CHECKING ASYNCHRONOUS SEQUENTIAL CIRCUITS

4.1 Asynchronous Machine Model

The model with which we will deal here is that of a standard asynchronous sequential machine operating in the fundamental mode, initially with single transition-time (STT) assignments. We will therefore assume single input changes. Operation of the machine is described by means of a flow table and we will assume that before this design process is begun the table has been reduced completely. The faults which are under consideration here are those faults which affect the internal mapping of inputs and present state into next states; we will not be concerned with faults in the output circuitry. Furthermore, it is assumed that, for the purpose of assuming single faults, and to check the checker, the machine will visit all of its internal states in a time less than the mean-time-between-failure (MTBF).

4.2 Internal Design for Self-Checking Characteristics

As the encoded output of the sequential machine is in the form of next-state values, any fault-detection capabilities must appear as changes in the next-state assignment. By avoiding redundancy in the next-state equations the circuit can be designed so that any fault will, for some present-state input, produce a next-state value which is different from normal. The difficult task is to assure fault-security, i.e. guarantee that the abnormal next state is not itself a valid codeword representing another state, but is rather a non-codeword. Therefore the set of error states (those state variable vectors produced by internal faults) and the set of proper next states must be disjoint sets. This is achieved by using a code such that the distance between codewords is greater than the distance between a given codeword and any of the error states to which it may be converted by any fault from the prescribed set.

Recall that normal design of an asynchronous machine requires that, for every pair of states between which there is a transition, there must be at least one state variable which partitions those states from the remainder of the states. To achieve the distance between codewords required for fault-security, Maki et al. [1] have shown that it is necessary and sufficient to provide two state variables which partition a given transition path from the others in the same input column. Then to assure that this distance is greater than that by which a codeword can be changed by a fault, the two partitioning variables must be implemented independently so that a single fault cannot affect them both. Finally, to prevent a transition from an error state to a valid state before the input changes again, and thus before the checker has had a chance to detect the illegal state, the next-state entries for the error states must be specified to be equal to their present-state values for the partitioning variables between transition paths. The operation of the machine is then such that any fault will cause the machine to enter an illegal state and remain there long enough to be detected. The state assignment procedure presented by Maki achieves this desired performance but it does not produce a state assignment which is compatible with known self-checking checkers. Hence the following procedure is presented which supplies the required partitioning and in addition yields a code, using as few state variables as possible, which may be checked by an easily designed self-checking checker.

Consider first the following essential definitions. Let S be the set of states of the machine, and let N_S be the number of states in S.

<u>Definition</u>: A <u>k-set</u> (Unger's [5] "destination set"), determined from an input column, is a subset of S consisting of a stable state and those states which have unstable entries in that column leading to that stable state.

<u>Definition</u>: A <u>π-partition</u> (Maki's [1] "column partition") is the collection of all the k-sets from a given column. If the column has no don't-care entries, then the π-partition will be completely specified, i.e. the union of the (disjoint) k-sets will be S. Part of the following procedure will convert all incompletely specified π-partitions to completely specified ones.

<u>Definition</u>: A <u>τ-partition</u> is a two-block partition of the members of S. Each π-partition will require one or more τ-partitions to cover it, i.e. for a given π-partition, it must be true that for every pair of k-sets in it, there exists a τ-partition that separates the pair into different blocks of that τ-partition. Each τ-partition will be completely specified over S and will ultimately determine two state variables, each of which partitions those states in one block from those in the other block.

4.3 <u>State Assignment Procedure</u>

<u>Goal</u>: Given a reduced flow table, produce a k-out-of-2k state assignment preserving fault-security.

<u>Step 1</u>: Starting with those columns containing no don't-care entries list all the π-partitions by listing all the k-sets from each column. To minimize the number of k-sets, note the following definition and theorem.

<u>Definition</u>: For a given input column, a <u>bachelor</u> <u>state</u> is a stable state with no transitions to it under that input.

<u>Theorem 1</u>: For a given input column, all bachelor states may be placed in the same k-set.

Proof: Let S_1 and S_2 be two bachelor states in a given input column. If S_1 and S_2 are in different k-sets under a different input then the required partitioning is taken care of in that π-partition, and there is no need to repartition S_1 from S_2. Thus they can be grouped together in the same k-set for this column. If S_1 and S_2 do not appear in different k-sets for any other input column then they are equivalent states and should be merged together and a new reduced flow table formed. This completes the proof.

Step 2: For a column containing an unspecified (don't-care) entry, first list the k-sets without including the state whose entry in that column is unspecified. Then add this state to one of these k-sets so that as many k-sets as possible in this π-partition match identically those in another π-partition. This merely helps minimize the number of state variables needed. Adding the state to a given k-set corresponds to placing in the don't-care entry an unstable entry of the stable state of that k-set.

Step 3: Delete any π-partition which contains only one k-set, or which is covered by another π-partition.

Step 4: For each remaining π-partition, determine a set of τ-partitions which cover it. Unger [4] and Tracey [5] offer techniques of determining minimal covers, or one may proceed according to the following method. Let n_k be the number of k-sets in a given π-partition. Then the number of τ-partitions required to cover π, n_τ, is given by

$$n_\tau = \lceil \log_2(n_k) \rceil$$

where $\lceil x \rceil$ means "the least integer not smaller than x." We will describe a technique to be used when $n_k = 2^j$ for some integer j. For $2^{j-1} < n_k < 2^j$

the technique is the same, with the excess $2^j - n_k$ k-sets disregarded.

Given 2^j k-sets in some arbitrary initial order. Generate the τ-partitions $\tau_1, \tau_2, \cdots$ in sequence as follows:

τ_1 has the first 2^{j-1} k-sets in its first block, the second 2^{j-1} k-sets in its second block.

τ_2 has the first 2^{j-2} k-sets from each block of τ_1 in its first block, the second 2^{j-2} k-sets from each block of τ_1 in its second block.

$\vdots$

τ_i has the first, third, $\cdots$, $(2i-3)^{rd}$ groups of 2^{j-i} k-sets from each block of τ_{i-1} in its first block, and the second, fourth, $\cdots$, $(2i-2)^{nd}$ groups of 2^{j-i} k-sets from each block of τ_{i-1} in its second block.

This process continues until $i = j$, since with $i = j$ we are now simply picking alternate k-sets from τ_{i-1} (group size $2^{j-i} = 1$). Thus the last τ-partition is τ_j, so $j = \log_2 2^j = \log_2(n_k)$ is the number of τ-partitions required to cover a π-partition containing $n_k = 2^j$ k-sets.

<u>Examples</u>: Let A,B,$\cdots$,P be generalized k-sets.

$n_k = 2$: $\tau = (A,B)$

$n_k = 3$: $\tau_1 = (AB,C)$ $\quad$ $\tau_2 = (AC,B)$

$n_k = 4$: $\tau_1 = (AB,CD)$ $\quad$ $\tau_2 = (AC,BD)$

$n_k = 5$: $\tau_1 = (ABC,DE)$ $\quad$ $\tau_2 = (ABD,CE)$ $\quad$ $\tau_3 = (ADE,BC)$

$n_k = 6$: $\tau_1 = (ABC,DEF)$ $\quad$ $\tau_2 = (ABD,CEF)$ $\quad$ $\tau_3 = (ADE,BCF)$

$n_k = 7$: $\tau_1 = (ABCD,EFG)$ $\quad$ $\tau_2 = (ABEF,CDG)$ $\quad$ $\tau_3 = (AECG,BFD)$

$n_k = 8$: $\tau_1 = (ABCD,EFGH)$ $\quad$ $\tau_2 = (ABEF,CDGH)$ $\quad$ $\tau_3 = (AECG,BFDH)$

$n_k = 16$: $\tau_1 = (\underline{ABCD}EFGH,\underline{IJKL}MNOP)$ $\quad$ $\tau_2 = (\underline{AB}CD\underline{IJ}KL,\underline{EF}GH\underline{MN}OP)$

$\tau_3 = (\underline{A}B\underline{I}J\underline{E}F\underline{M}N,\underline{C}D\underline{K}L\underline{G}H\underline{O}P)$ $\quad$ $\tau_4 = (AIEMCKGO,BJFNDLHP)$.

In the last example, the underlined k-sets in τ_i, i=1,2,3 are those that go in the first block of τ_{i+1}. Note that in every example, any two k-sets appear in different blocks of at least one τ-partition.

For $n_k > 2$, the above choices are only suggested, not unique, and choices should be made wherever possible to find τ-partitions which help cover more than one π-partition. An example of this follows.

Example: Given S = $\{a,b,c,d,e\}$ and the following set of π-partitions:

π_1 = (ab,cde)

π_2 = (ade,bc)

π_3 = (ae,bcd)

π_4 = (a,bc,de)

π_1 is covered by τ_1 = (ab,cde)

π_2 is covered by τ_2 = (ade,bc)

π_3 is covered by τ_3 = (ae,bcd)

π_4 is covered by any two of the following three τ-partitions:

τ_4 = (a,bcde)

τ_5 = (abc,de)

τ_6 = (ade,bc) .

It is immediately seen that $\tau_6 = \tau_2$ so τ_2 can be used to partition <u>a</u> from <u>bc</u> and <u>de</u> from <u>bc</u> (all derived from π_4). All that is needed then is a τ-partition which partitions <u>a</u> from <u>de</u>. This is already done by τ_1, so in this example τ_1, τ_2, and τ_3 cover all four π-partitions.

<u>Step 5</u>: Suppose we now have all π-partitions covered and have N_τ unique τ-partitions. We now generate $N_V = 2 \cdot N_\tau$ state variables by associating two unique state variables with each τ-partition as follows:

$Y_i = 1$ for all states in the left block of τ_i,

$Y_i = 0$ for all states in the right block of τ_i,

$Y_{N_\tau+i} = \overline{Y_i}$, $i=1,\cdots,N_\tau$.

For those columns producing only two k-sets, this method produces the same state variables as Maki's [1] method since, although this method requires two state variables per τ-partition, Maki's technique would have two τ-partitions for each two-k-set π-partition. In general, Maki requires only one state variable per τ-partition since he produces one τ-partition per unique k-set; this method yields two state variables per τ-partition, but requires fewer τ-partitions to cover all the k-sets in their respective π-partitions.

With the Y's in order from Y_1 to Y_{2N_τ} this yields a k-out-of-2k state assignment with $k = N_\tau$. Furthermore, the state assignment is such that each state vector can be divided into two groups A and B containing equal numbers of bits with $b_i = \overline{a_i}$ where $a_i = Y_i$ and $b_i = Y_{N_\tau+i}$, $i=1,\cdots,k$. Finally, each k-set is partitioned from every other k-set in its π-partition by two state variables.

A bound on the number of state variables can be computed directly from N_π, the total number of π-partitions, and n_{k_i}, the number of k-sets in π_i, $i=1,\cdots,N_\pi$, and is given by

$$N_V \leq \sum_{i=1}^{N_\pi} 2\cdot\left\lceil \log_2(n_{k_i}) \right\rceil$$

with equality occurring in the case where no τ-partition helps cover more than one π-partition (i.e. when the mapping of τ-partitions onto π-partitions is one-to-one).

4.4 Examples

We will now illustrate the state assignment procedure with several examples.

Example A: Consider the flow table in Figure 1. List the π-partitions:

$\pi_1 = (ac,bd)$

$\pi_2 = (abcd)$

$\pi_3 = (ad,bc)$

$\pi_4 = (a,bc,d) = (ad,bc)$ from Theorem 1.

π_4 and π_2 can now be eliminated according to Step 3.

Determine τ-partitions:

$\tau_1 = (ac,bd)$

$\tau_2 = (ad,bc)$.

Assign state variables:

	Y_1	Y_2	Y_3	Y_4
a =	1	1	0	0
b =	0	0	1	1
c =	1	0	0	1
d =	0	1	1	0 .

Example B: Consider the flow table in Figure 2, which was adapted from Maki's machine A by merging states b and c and adding an additional input column.

List π-partitions:

$\pi_1 = (ae,cd)$ from columns 00 and 01

$\pi_2 = (ac,de)$ from column 11

$\pi_3 = (ad,ce)$ from column 10 .

x_1x_2

	00	01	11	10
a	ⓐ	b	d	ⓐ
b	ⓑ	ⓑ	c	c
c	a	b	ⓒ	ⓒ
d	b	b	ⓓ	ⓓ

Figure 1. Flow table for example A.

x_1x_2

	00	01	11	10
a	ⓐ	e	ⓐ	ⓐ
c	ⓒ	ⓒ	a	ⓒ
d	-	c	ⓓ	a
e	a	ⓔ	d	c

Figure 2. Flow table for example B.

List τ-partitions:

τ_1 = (ae,cd)

τ_2 = (ac,de)

τ_3 = (ad,ce) .

Assign state variables:

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
a =	1	1	1	0	0	0
c =	0	1	0	1	0	1
d =	0	0	1	1	1	0
e =	1	0	0	0	1	1 .

Example C: Consider the flow table in Figure 3.

List the π-partitions:

π_1 = (ae,bcd) from columns 00 and 01

π_2 = (ac,b,de) from column 11

π_3 = (ad,b,ce) from column 10 .

List the τ-partitions:

τ_1 = (ae,bcd) covers π_1

τ_2 = (abc,de) helps cover π_2

τ_3 = (ad,bce) helps cover π_3

τ_4 = (b,acde) helps cover π_2 and π_3 .

Here the maximum number of τ-partitions required to cover the three π-partitions (five) has been reduced to four through the sharing of one τ-partition.

	x_1x_2			
	00	01	11	10
a	ⓐ	e	ⓐ	ⓐ
b	ⓑ	c	ⓑ	ⓑ
c	b	ⓒ	a	ⓒ
d	-	c	ⓓ	a
e	a	ⓔ	d	c

Figure 3. Flow table for example C.

Assign state variables:

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8
a =	1	1	1	0	0	0	0	1
b =	0	1	0	1	1	0	1	0
c =	0	1	0	0	1	0	1	1
d =	0	0	1	0	1	1	0	1
e =	1	0	0	0	0	1	1	1 .

When attaching this to a self-checking checker, the checker's output functions f_4 and g_4 would be implemented according to the design equations presented earlier, with $a_1 \cdots a_4 = Y_1 \cdots Y_4$ and $b_1 \cdots b_4 = Y_5 \cdots Y_8$.

5. ADAPTING THE SEQUENTIAL MACHINE TO THE CHECKER

5.1 General Problems

Having now achieved a state assignment employing a code for which we can easily design a self-checking checker, we must now attack the problem of implementing the checker. According to the design equations for a k-out-of-2k checker given earlier, any realization of the functions f_k and g_k will yield a self-checking checker. However, a two-level realization requires the presentation of all the codewords in the code space to the checker to check for all of its possible internal faults. Since our codewords are states of a finite state machine, we have available only those codewords which are assigned to states of the machine. In most cases this is far fewer than the number required to check the checker.

To overcome this, we can start by choosing a realization of the checker that will enable it to be checked by fewer than the $\binom{2k}{k}$ codewords that make up the code space. Anderson [2] showed that when the checker is implemented directly from the equations, resulting in a four-level realization, it can be checked using only a special set of 2^k codewords, namely those in which the rightmost k bits are the bitwise complements of the leftmost k bits, i.e. $y_{k+i} = \bar{y}_i$, $i=1,\cdots,k$. The state assignment procedure just presented does in fact yield codewords with this property, so we are therefore lacking only 2^k-N_S codewords, rather than $\binom{2k}{k} - N_S$ codewords. Note that of the three examples presented above, only example A has sufficient codewords on its own.

This problem of insufficient codewords is a serious one and cannot be ignored, for if certain faults in the checker go undetected because we do not have the codewords to check for them, then our single fault assumption is no longer valid. With an undetectable fault in existence, the occurrence of another fault, which ordinarily might be detectable, may produce a situation whereby the two faults combine to destroy the checking properties of the checker. Most notably, they could destroy the code-disjoint property, and cause a non-code input to yield a code-space output. Since preventing inaccurate transmission of data of this nature is one of the primary values of having a real-time self-checking network, to allow such an error to occur would negate any value the checker adds to the confidence of the network.

5.2 Methods Which Do Not Work

Several methods of attacking the problem of insufficient codewords were considered. One idea was to use only the available codewords and eliminate those portions of the checker which check for the occurrence of the unavailable codewords and which are themselves checked for internal faults by the same unavailable codewords. This method was found not to work, however, since by eliminating those portions of the checker which check for the occurrence of the codewords we are not using, we lose the power to check for increasing (zero-to-one) faults in the codewords we are using. Thus the checker must remain intact, and the only solution lies in somehow making all the necessary codewords available to the checker.

The idea of using a special test input which when switched on would feed the proper codewords onto the checker input lines was considered but also rejected because the test input must itself be testable, and the addition of the logic it requires makes that too difficult to be practical.

A third consideration, also ultimately rejected, was to use the method of multicode STT assignments presented in section 3.3.4 of Unger [5]. This technique, commonly called state-splitting or row-splitting, assigns several different codewords to a given state, the maximum number being the number of columns in which that state is stable. Hence if a state A is stable in three columns, it may be represented by A_1, A_2, and A_3. As is shown in table 3.16c in Unger [5], A_1, A_2, and A_3 would then be stable in every column in which A was originally stable, with transitions to A in the first column in which it was stable now going to A_1, and similarly for A_2 and A_3. Actually, these transitions need not be so constrained, and can in fact go to any A_i, but having all those in a given column go to the same A_i means that the other split versions of A are bachelor states in that column, and hence at most one new k-set must be added to the existing k-sets from that column. The flow table now has a number of new stable states added to it, and even though the additional states in each column are bachelor states, this leads to problems of partitioning.

To understand this, consider the situation from an abstract point of view. We started with an original flow table which produced a set of partitions and a state assignment. This assignment defined a code space and, along with the codewords assigned to existing states, a number of new codewords which must have states assigned to them. This additional assignment implies an expanded flow table and essentially a new machine, which is subject to two major constaints. First, it must operate the same as the old machine in mapping input sequences into outputs. And second, it must be such that if its flow table were simply taken as an original flow table, it must yield a state assignment identical to the one which has already been specified.

In light of these constraints, how can these split states be partitioned to preserve the essential properties? If the split states are considered equivalent by virtue of their having been derived from the same original unsplit counterpart, then they need not be partitioned from each other. Thus they will all be assigned the same state assignment and will be of no use in producing new codewords. If, on the other hand, they are considered independent and partitioned like any other stable states, then in almost every case this new partitioning will require a new state assignment code of the form (k+j)-out-of-2(k+j) for some non-zero j equalling the number of new τ-partitions. This new code will in turn require more state-splitting to supply even more codewords. This will not converge to a feasible assignment.

The only case in which new partitioning does not require a new state assignment code is the case where each column produces its own unique π-partition, no π-partition contains a number of k-sets that is a power of two, and no τ-partition helps cover more than one π-partition. However, even in this case state-splitting will not work because the number of new states created by splitting every state that is stable in more than one column into ultimately as many states as columns in which it is stable will still be less than the number of new codewords required to reach 2^k. This is verified by the following argument.

Let m be the number of input columns and x be the number of states of the machine. m and x are both positive integers, and for the sake of simplicity and maximum state splitting assume they are both even numbers. With maximum state splitting, the maximum number of total states is $\frac{mx}{2}$, and the number of τ-partitions is $m \cdot \left\lceil \log_2 (\frac{x}{2}) \right\rceil$. Therefore to have at least as many total states as codewords required, it must be true that

$$\frac{mx}{2} > 2^{m \cdot \left\lceil \log_2 (\frac{x}{2}) \right\rceil}$$

which implies $\log_2(\frac{mx}{2}) \geq m\left\lceil \log_2(\frac{x}{2}) \right\rceil$.

Since $\left\lceil \log_2(\frac{x}{2}) \right\rceil \geq \log_2(\frac{x}{2})$, then the above implies $\log_2(\frac{mx}{2}) \geq m \cdot \log_2(\frac{x}{2})$ which implies $\log_2(\frac{mx}{2}) \geq \log_2(\frac{x}{2})^m$ and hence $(\frac{x}{2}) \cdot m \geq (\frac{x}{2})^m$.

With the above constraints on m, x, and the machine (for example $x > 4$ since $x \leq 4$ means that every π-partition will have n_k equalling a power of two), the final inequality cannot possibly occur, and therefore state-splitting will not work. Hence the following method seems to be the only valid technique for acquiring additional codewords.

This technique involves the assignment of the additional codewords as unstable states which may be cycled through during transitions between the machine's original stable states. This allows the partitioning dictated by the 2^k codewords to stand unimpaired, as transitions in a given column may pass through any states having a value of 1 for the state variable originally created by the τ-partition for those transitions in that column. It is assumed, of course, that the time during which the machine is in these unstable states is sufficient to check for the corresponding faults in the checker for which the new codewords are required. Note that the machine is no longer operating with STT assignments.

5.3 <u>Technique for Cycling Through Additional States</u>

<u>Step 1</u>: List the additional codewords underneath the original ones in any order.

<u>Step 2</u>: Label the new codewords with state names similar to the original state names.

<u>Step 3</u>: Now, paying attention only to the leftmost k columns in the state assignment, corresponding to $Y_1 \cdots Y_k$, recall that each column was determined by a unique τ-partition. The new states must be added to the τ-partitions

so that if the τ-partitions with these additions were taken a priori then they would produce the state assignments which we now have and cannot change. Therefore add those new states that have 1-entries in column i (i.e. Y_i) to the left block of τ_i, and those that have 0-entries to the right block of τ_i, $i=1,\cdots,k$. When this has been completed for all k τ-partitions, each τ-partition will now have 2^{k-1} states in each block, regardless of how many it had in each block initially.

Step 4: The new states in a given block of a given τ-partition may now be cycled through as unstable states in the transitions that occur among the original states of that block, provided such transitions exist. Blocks originally containing only bachelor states cannot be used, nor can implied transitions caused by a don't-care entry in the flow table. Blocks containing valid transitions will be called "viable blocks".

Example: Suppose a block originally containing S_1 and S_2 (among others) now has S_{11} and S_{12} added. If the original transition was S_2-S_1, it may be replaced by S_2-S_{11}-S_{12}-S_1.

To achieve the greatest speed, it may be desirable to include each new state in only one transition. In the above example, if S_{12} is in a viable block in another τ-partition then it may be used in a transition there, and the above transition may be reduced to S_2-S_{11}-S_1. Referring to the original flow table and the new τ-partitions, include each new state in some legitimate transition. This dispersal of the new states throughout the transitions of the original machine requires the assumption that under normal operation all of these transitions will occur, and in a time period less than the assumed MTBF. In cases where certain transitions are known to occur more frequently than others, it may be wise to include as many new states as possible in those transitions.

<u>Step 5</u>: The flow table must be modified to reflect the addition of the new states. For a given column, those transitions which are to be replaced by sequences of transitions involving the new unstable states are modified by placing in the row of each state in the sequence an unstable entry of the next state in the sequence. Those states which are not involved in transitions under that input may have don't-care entries placed in the table under that input. No stable state entries are added to or deleted from the original stable entries in the original flow table.

5.4 Examples

To illustrate this procedure, we will continue with the three examples which served to illustrate the state assignment procedure.

<u>Example A</u>: The state assignment for the machine whose flow table appears in Figure 1 was shown to be a 2-out-of-4 code. This requires $2^2 = 4$ codewords. Since the machine has four states and their code is such that $Y_{k+i} = \overline{Y}_i$ no additional codewords are needed to check the checker.

<u>Example B</u>: The state assignment for machine B whose flow table appears in Figure 2 is a 3-out-of-6 code and thus requires $2^3 = 8$ codewords, of which we have only four. The additional four codewords we need are the four remaining binary combinations of $Y_1Y_2Y_3$.

Assign and label new codewords:

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
a =	1	1	1	0	0	0
c =	0	1	0	1	0	1
d =	0	0	1	1	1	0
e =	1	0	0	0	1	1

f =	1	1	0	0	0	1
g =	1	0	1	0	1	0
h =	0	1	1	1	0	0
i =	0	0	0	1	1	1 .

Add new states to τ-partitions:

$$\tau_1 = (aefg, cdhi)$$

$$\tau_2 = (acfh, degi)$$

$$\tau_3 = (adgh, cefi) \ .$$

Note for example that $Y_2 = 1$ for those states in the left block of τ_2.

Modify flow table: The first modification of the flow table, illustrated in Figure 4, uses all the new states wherever a transition is available. This maximally specified flow table produces a slow but sure machine. Looking under column 00 for instance, one may see that the original e-a transition has been replaced by e-f-g-a, as is indicated by the first block of τ_1. The second modification of the flow table for machine B, illustrated in Figure 5, uses each new state in only one transition. For machines whose transitions occur with uniform frequency this minimal specification is the more efficient method.

Example C: The state assignment for machine C, whose flow table is illustrated in Figure 3, is a 4-out-of-8 code requiring $2^4 = 16$ codewords to check the checker. Hence we must add eleven new states.

Assign and label new codewords:

	x_1x_2			
	00	01	11	10
a	$\textcircled{a}$	f	$\textcircled{a}$	$\textcircled{a}$
c	$\textcircled{c}$	$\textcircled{c}$	f	$\textcircled{c}$
d	-	h	$\textcircled{d}$	g
e	f	$\textcircled{e}$	g	f
f	g	g	h	i
g	a	e	i	h
h	-	i	a	a
i	-	c	d	c

Figure 4. Maximally specified expanded flow table for example B.

	x_1x_2			
	00	01	11	10
a	$\textcircled{a}$	g	$\textcircled{a}$	$\textcircled{a}$
c	$\textcircled{c}$	$\textcircled{c}$	h	$\textcircled{c}$
d	-	c	$\textcircled{d}$	a
e	f	$\textcircled{e}$	d	i
f	a	-	-	-
g	-	e	-	-
h	-	-	a	-
i	-	-	-	c

Figure 5. Minimally specified expanded flow table for example B.

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8
a =	1	1	1	0	0	0	0	1
b =	0	1	0	1	1	0	1	0
c =	0	1	0	0	1	0	1	1
d =	0	0	1	0	1	1	0	1
e =	1	0	0	0	0	1	1	1
f =	0	0	0	1	1	1	1	0
g =	1	1	0	0	0	0	1	1
h =	1	0	1	0	0	1	0	1
i =	1	0	0	1	0	1	1	0
j =	0	1	1	0	1	0	0	1
k =	0	0	1	1	1	1	0	0
ℓ =	1	1	1	1	0	0	0	0
m =	1	1	0	1	0	0	1	0
n =	1	0	1	1	0	1	0	0
o =	0	1	1	1	1	0	0	0
p =	0	0	0	0	1	1	1	1 .

Add new states to τ-partitions:

τ_1 = (aeghiℓmn,bcdfjkop)

τ_2 = (abcgjℓmo,defhiknp)

τ_3 = (adhjkℓno,bcefgimp)

τ_4 = (bfikℓmno,acdeghjp) .

Note that the first block of τ_4 is not a viable block and therefore cannot be used to cycle through any new states.

Modify flow table: The maximally specified flow table is illustrated in Figure 6; the minimally specified flow table is illustrated in Figure 7.

	x_1x_2			
	00	01	11	10
a	ⓐ	g	ⓐ	ⓐ
b	ⓑ	f	ⓑ	ⓑ
c	f	ⓒ	g	ⓒ
d	-	o	ⓓ	h
e	g	ⓔ	f	f
f	j	j	h	g
g	h	h	j	i
h	i	i	i	j
i	ℓ	ℓ	k	m
j	k	k	ℓ	k
k	o	c	n	ℓ
ℓ	m	m	m	n
m	n	n	o	p
n	a	e	p	o
o	p	p	a	a
p	b	c	d	c

Figure 6. Maximally specified expanded flow table for example C.

	x_1x_2			
	00	01	11	10
a	$\textcircled{a}$	i	$\textcircled{a}$	$\textcircled{a}$
b	$\textcircled{b}$	k	$\textcircled{b}$	$\textcircled{b}$
c	f	$\textcircled{c}$	ℓ	c
d	-	0	$\textcircled{d}$	m
e	g	$\textcircled{e}$	n	p
f	j	-	-	-
g	h	-	-	-
h	a	-	-	-
i	-	e	-	-
j	b	-	-	-
k	-	c	-	-
ℓ	-	-	a	-
m	-	-	-	a
n	-	-	d	-
o	-	c	-	-
p	-	-	-	c

Figure 7. Minimally specified expanded flow table for example C.

6. DESIGN EQUATION CONSIDERATIONS

The only remaining aspect of the design process is the generation of the design equations for the state variables. For those machines which have 2^k original stable states and generate a k-out-of-2k state assignment, such as in example A, Maki's short-cut method of producing design equations by inspection may be employed. For those machines requiring the addition of unstable states to supply additional codewords, conventional methods may be used. As that is simply a mechanical exercise, it will not be illustrated here. In either case, for any Y_i, $i=1,\cdots,2k$, the term $y_i I_j$ must be included or covered, where I_j is the input column (or the OR of the input columns) which determined the τ-partition from which Y_i was generated. This assures that during a transition any error-free partitioning variable will retain its present value so that if the other partitioning variable is affected by a fault the machine will assume an error state and remain in one at least until the input changes. Finally, whatever realization is chosen, the next-state variables must be independently implemented from the inputs and present-state variables. This is absolutely essential to maintain fault-security.

7. CONCLUDING REMARKS

A design method has been presented here for the construction of totally self-checking asynchronous sequential machines. These machines include a check circuit which not only detects errors in the state-determination logic of the sequential portion but which detects errors in its own internal circuitry as well. All checking of the entire system is therefore done in real time, thus obviating interruption of normal operation simply to check for faults. The necessary state assignments are easily generated and the remainder of the design procedure is rarely more difficult, and sometimes easier, than standard techniques. The price of the self-checking property is sometimes paid for in logic, and the determination of whether this feature is worth the price is ultimately left up to the individual designer.

LIST OF REFERENCES

1. D. W. Sawin, III, G. K. Maki, and S. R. Groenig, "Design of Asynchronous Sequential Machines for Fault Detection," Digest 1972 International Symposium on Fault-Tolerant Computing, Newton, Mass., pp. 170-175, June 1972.

2. D. A. Anderson, "Design of Self-Checking Digital Networks Using Coding Techniques," CSL Report R-527, University of Illinois, Ph.D. Thesis, October 1971.

3. D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," IEEE Trans. on Computers, Vol. C-22, pp. 263-269, March 1973.

4. S. H. Unger, Asynchronous Sequential Switching Circuits, Wiley-Interscience, N.Y., 1969.

5. J. H. Tracey, "Internal State Assignments for Asynchronous Sequential Machines," IEEE Trans. on Electronic Computers, Vol. EC-15, pp. 551-556, August 1966.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-73-593	2.	3. Recipient's Accession No.
4. Title and Subtitle Design of Totally Self-Checking Asynchronous Sequential Machines			5. Report Date September 1973
			6.
7. Author(s) Daniel Avery Pitt			8. Performing Organization Rept. No. UIUCDCS-R-73-593
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			13. Type of Report & Period Covered Master of Science Thesis
			14.

15. Supplementary Notes

16. Abstracts

Techniques have been developed for designing asynchronous machines that indicate faults by assuming error states. Design methods have also been found for the construction of self-checking combinational networks which indicate errors either in their coded inputs or in their own circuitry. This paper presents a means of uniting these two methods to yield asynchronous sequential machines which are self-checking. The problems inherent in the individual design procedures for the two methods as well as those which arise in making the two methods compatible are discussed.

17. Key Words and Document Analysis. 17a. Descriptors

Fault-detection
Sequential machines
Self-checking

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Umlimited Release	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 37
	20. Security Class (This Page UNCLASSIFIED	22. Price -----